Java Editions:

J2SE (Java Standard Edition)

Core Java,OOps,Threads,Exceptions, Swings(To develop stand alone).

J2EE (Java Enterprise Edition)

Servlets, JSP, Spring, Spring Boot (To develop client server application)

J2ME (Java Mobile/Micro Edition)

To develop Mobile editions

JDK, JRE, JVM

JDK (JRE + Development Tools):

****

To develop, compile and run the Java applications. Platform dependent.

JRE (JVM + Class Libraries):

****

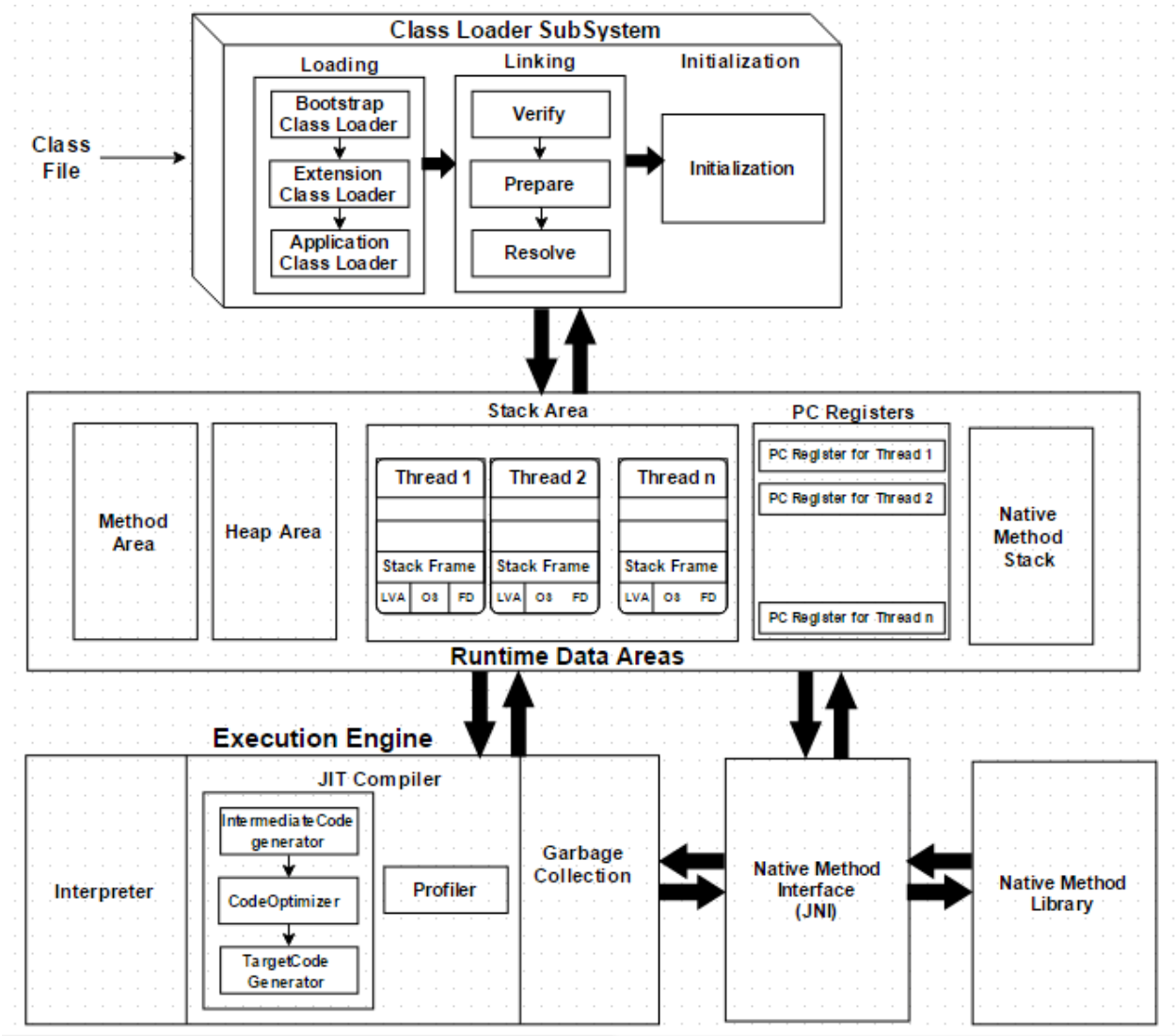
To Run the Java applications. Platform dependent.

JVM (Responsible for load and run java class files):

****

Its an interpreter to execute the Java programs (Byte code) line by line and convert it to OS specific code. Platform independent.

JVM internally contain below things init



1. Class Loader Sub System(Load the .class file)

****

Class Loader Stages

****

Loading

Responsible to read the .class from hard disk and load it to Method area

Different types class loaders

****

- Boot Strap Class Loader (Responsible for loading rt.jar(Internal Classes)),
- Extesion Class Loader (Responsible loading remaining jars available in JRE folder (jre/lib/ext)),
- Application Class Loader (Responsible to load the class path files)

Linking

Verification: It ensures the correctness of the .class file i.e. it checks whether this file is properly formatted and generated by a valid compiler or not. If verification fails, we get run-time exception java.lang.VerifyError. This activity is done by the component ByteCodeVerifier. Once this activity is completed then the class file is ready for compilation.

Preparation: JVM allocates memory for class variables and initializing the memory to default values.

Resolution: It is the process of replacing symbolic references from the type with direct references. It is done by searching into the method area to locate the referenced entity.

Initialization

In this phase, all static variables are assigned with their values defined in the code and static block(if any). This is executed from top to bottom in a class and from parent to child in the class hierarchy.

2. Runtime Data Areas (Store the data)

****

Method Area (Stores Class level info(Methods Info, Varibales Info, Constructor Info,Modifiers Info,Constant Pool Info .. etc))

Heap Area (Stores Objects(Class Class Object Info) Info)

Stack Area (Stores Local Variables)

Pc Register Area (Threads info)

Native Method Stack Area (Native method stack info)

3. Execution Engine (Execute the Program)

****

1. Interpreter

It interprets the bytecode line by line and then executes. The disadvantage here is that when one method is called multiple times, every time interpretation is required.

2. JIT Compiler

It is used to increase the efficiency of an interpreter. It compiles the entire bytecode and changes it to native code so whenever the interpreter sees repeated method calls, JIT provides direct native code for that part so re-interpretation is not required, and thus efficiency is improved.

3. Garbage Collector

It destroys un-referenced objects. For more on Garbage Collector, refer

Native Method Libraries:

It is a collection of the Native Libraries(C, C++) which are required by the Execution Engine.

Java Native Interface (JNI):

****

It is an interface that interacts with the Native Method Libraries and provides the native libraries(C, C++) required for the execution. It enables JVM to call C/C++ libraries and to be called by C/C++ libraries which may be specific to hardware.

Reference: https://www.geeksforgeeks.org/jvm-works-jvm-architecture/

****

Garbage Collection

****

Garbage Collector is responsible for garbage collection.

Garbage collector is best example of Daemon thread as it is always running in background. Main objective of Garbage Collector is to free heap memory by destroying unreachable objects.

Heap divided into three parts:

Young Generation:

Newly created objects start in the Young Generation. If young space is full then garbage collector will be called and memory will be released.

Old Generation:

Objects that are long-lived are eventually moved from the Young Generation to the Old Generation. When objects are garbage collected from the Old Generation, it is a major garbage collection event.

Permanent Generation:

Metadata such as classes and methods are stored in the Permanent Generation. Classes that are no longer in use may be garbage collected from the Permanent Generation.

Garbage Collector can be called manually by calling below,

System.gc()

Runtime.gc()

Above two was not recommended and no guarantee of releasing the memory.

Finalization:

Just before destroying an object, Garbage Collector calls finalize() method on the object to perform cleanup activities.

Once finalize() method completes, Garbage Collector destroys that object. The finalize() method called by Garbage Collector not JVM. Although Garbage Collector is one of the module of JVM.

Ways to make an object eligible for garbage collection in Java-:

By nullifying the reference variable

```
Student s1 = new Student( );
Student s2 = new Student( );
s1 = null;
s2 = null;
```

By reassigning the reference variable

```
Student s1 = new Student( );
Student s2 = new Student( );

//Not eligible for garbage collections
s1 = new Student( );
s2=s1
```

By creating objects inside a method

```
class Test
{
 public static void main(String[ ] args)
 {
    m1( );
 }

 public static void m1( )
 {
```

```
    Student s1 = new Student( );
    Student s2 = new Student( );
 }
}
```

Reference: https://www.geeksforgeeks.org/garbage-collection-java/

https://dzone.com/articles/jvm-architecture-explained

OOPS

Class:

Its a collections of objects/ Blue print of object. Its a kind of drawing an object.

Object:

Its a real world entity. It has State and behavior.

Ex: Class Dog

Properties: Size, Breed, age, color

Behaviour: eat(), sleep (), run (), bark()

Abstraction:

Its a process of hiding the internal details and showing the functionality. Fro example Car.

Encapsulation:

Wrapping up the data into an single unit. As in encapsulation, the data in a class is hidden from other classes, so it is also known as data-hiding. Example Capsule.

Inheritance:

Acquiring all the properties and behaviors of a parent object

Polymorphism:

Ability of having more than one form. Example Method Overloading and overriding.

There are two types of polymorphism in Java: compile-time polymorphism and runtime polymorphism. We can perform polymorphism in java by method overloading and method overriding.

If you overload a static method in Java, it is the example of compile time polymorphism. Here, we will focus on runtime polymorphism in java.

Runtime polymorphism or Dynamic Method Dispatch is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

https://beginnersbook.com/2013/03/polymorphism-in-java/

Out Memory Error

This error is thrown when there is insufficient space to allocate an object in the Java heap. In this case, The garbage collector cannot make space available to accommodate a new object, and the heap cannot be expanded further.

Steps to avoid it:

----

1. Nullify the object references.

2. Close the Databse connections

3. Increase the Heap area size

4. Take care of loops.

5. Take care of array size declaration

Strings:

Any array of character is String.

How to create a string object?

There are two ways to create String object:

By string literal

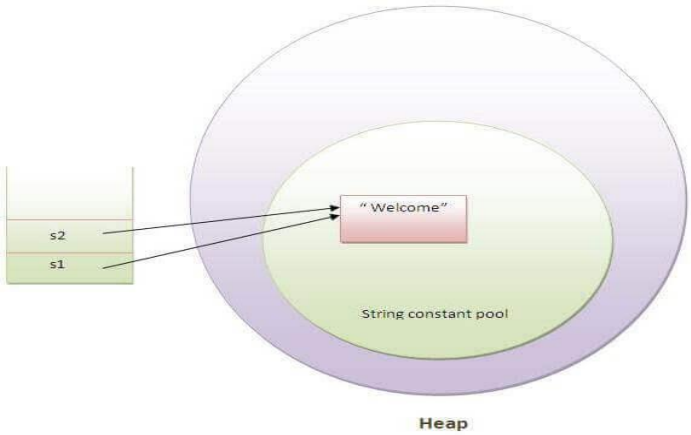
Java String literal is created by using double quotes. For Example:

String s="welcome";

Each time you create a string literal, the JVM checks the "string constant pool" first. If the string already exists in the pool, a reference to the pooled instance is returned. If the string doesn't exist in the pool, a new string instance is created and placed in the pool. For example:

String s1="Welcome";

String s2="Welcome";//It doesn't create a new instance



In the above example, only one object will be created. Firstly, JVM will not find any string object with the value "Welcome" in string constant pool, that is why it will create a new object. After that it will find the string with the value "Welcome" in the pool, it will not create a new object but will return the reference to the same instance.

By new keyword

String s=new String("Welcome**");//creates two objects and one reference variable**

In such case, JVM will create a new string object in normal (non-pool) heap memory, and the literal "Welcome" will be placed in the string constant pool. The variable s will refer to the object in a heap (non-pool).

Methods:

****

equals() => Compare the content of objects

Abstract Classes & Interface

Abstract Classes

A class which is declared with the abstract keyword is known as an abstract class in Java. It can have abstract and non-abstract methods (method with the body).

1. It can not be instantiated

2. It can have final methods

3. Abstart class can be decleared with abstract keyword

4. It can have both abstaract and non abstract methods

5. It can have Constructors and Static Methods

Uses:

----

1. Code reusability

2. To achieve loose coupling

Interfaces:

An interface is a collection of abstract methods

The interface in Java is a mechanism to achieve abstraction. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java.

1. To support multiple inheritance

2. It does not have constructor.

3. It does not have access modifiers.

4. We can acheive loose coupling

5. We can not initiate interface.

Collections:

Group of individual objects which are represented as a single unit is known as the collection of the object

What is a Framework?

A framework is a set of classes and interfaces which provide a ready-made architecture.

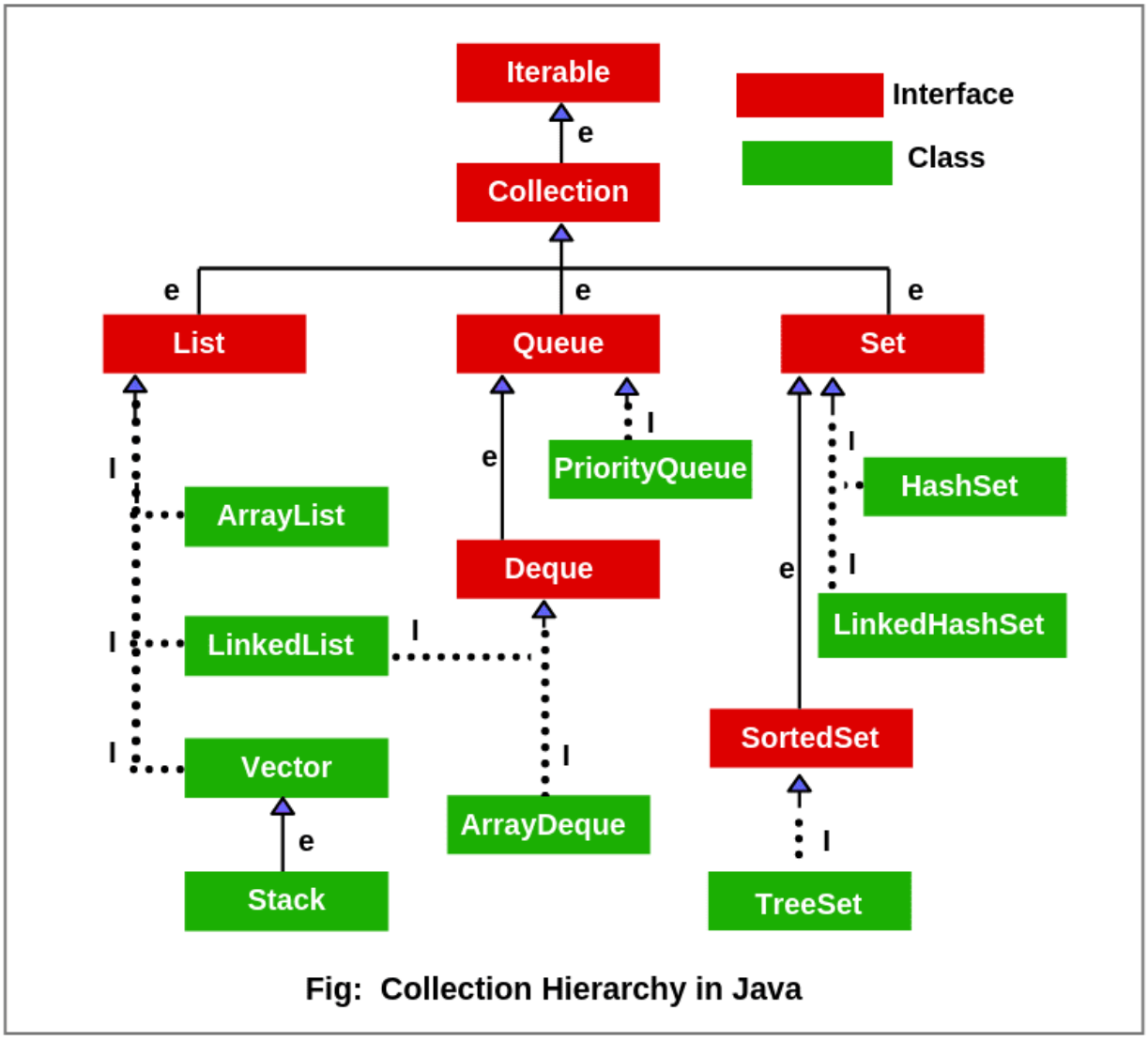


Fig: Collection Hierarchy in Java

Iterable Interface:

This is the root interface for the entire collection framework. The collection interface extends the iterable interface. Therefore, inherently, all the interfaces and classes implement this interface. The

main functionality of this interface is to provide an iterator for the collections. Therefore, this interface contains only one abstract method which is the iterator.

Collection Interface:

This interface extends the iterable interface and is implemented by all the classes in the collection framework. This interface contains all the basic methods which every collection has like adding the data into the collection, removing the data, clearing the data,

List:

ArrayList:

ArrayList provides us with dynamic arrays in Java. Though, it may be slower than standard arrays but can be helpful in programs where lots of manipulation in the array is needed. The size of an ArrayList is increased automatically

Methods: add, addAll, remove, set, clear,contains

LinkedList

LinkedList class is an implementation of the LinkedList data structure which is a linear data structure where the elements are not stored in contiguous locations and every element is a separate object with a data part and address part. The elements are linked using pointers and addresses.

Vector

A vector provides us with dynamic arrays in Java. Though, it may be slower than standard arrays but can be helpful in programs where lots of manipulation in the array is needed. This is identical to ArrayList in terms of implementation. However, the primary difference between a vector and an ArrayList is that a Vector is synchronized and an ArrayList is non-synchronized

Stack

Stack class models and implements the Stack data structure. The class is based on the basic principle of last-in-first-out. In addition to the basic push and pop operations, the class provides three more functions of empty, search and peek

Set:

A set is an unordered collection of objects in which duplicate values cannot be stored. This collection is used when we wish to avoid the duplication of the objects and wish to store only the unique objects

HashSet:

The HashSet class is an inherent implementation of the hash table data structure. The objects that we insert into the HashSet do not guarantee to be inserted in the same order. The objects are inserted based on their hashcode. This class also allows the insertion of NULL elements.

LinkedHashSet:

A LinkedHashSet is very similar to a HashSet. The difference is that this uses a doubly linked list to store the data and retains the ordering of the elements.

SortedSet:

This interface is very similar to the set interface. The only difference is that this interface has extra methods that maintain the ordering of the elements. The sorted set interface extends the set interface and is used to handle the data which needs to be sorted.

TreeSet:

The TreeSet class uses a Tree for storage. The ordering of the elements is maintained by a set using their natural ordering whether or not an explicit comparator is provided. This must be consistent with equals if it is to correctly implement the Set interface.

Queue:

As the name suggests, a queue interface maintains the FIFO(First In First Out) order similar to a real-world queue line. This interface is dedicated to storing all the elements where the order of the elements matter. For example, whenever we try to book a ticket, the tickets are sold at the first come first serve basis. Therefore, the person whose request arrives first into the queue gets the ticket.

Priority Queue:

A PriorityQueue is used when the objects are supposed to be processed based on the priority. It is known that a queue follows the First-In-First-Out algorithm, but sometimes the elements of the queue are needed to be processed according to the priority and this class is used in these cases. The PriorityQueue is based on the priority heap. The elements of the priority queue are ordered according to the natural ordering, or by a Comparator provided at queue construction time, depending on which constructor is used.

DeQueue

This is a very slight variation of the queue data structure. Deque, also known as a double-ended queue, is a data structure where we can add and remove the elements from both the ends of the queue.

ArrayQueue

ArrayDeque class which is implemented in the collection framework provides us with a way to apply resizable-array. This is a special kind of array that grows and allows users to add or remove an element from both sides of the queue. Array deques have no capacity restrictions and they grow as necessary to support usage.

Map:

A map is a data structure which supports the key-value pair mapping for the data. This interface doesn't support duplicate keys because the same key cannot have multiple mappings

HashMap

HashMap is the implementation of Map, but it doesn't maintain any order.

LinkedHashMap:

LinkedHashMap is the implementation of Map. It inherits HashMap class. It maintains insertion order.

HashTable

Java Hashtable class implements a hashtable, which maps keys to values. It inherits Dictionary class and implements the Map interface.

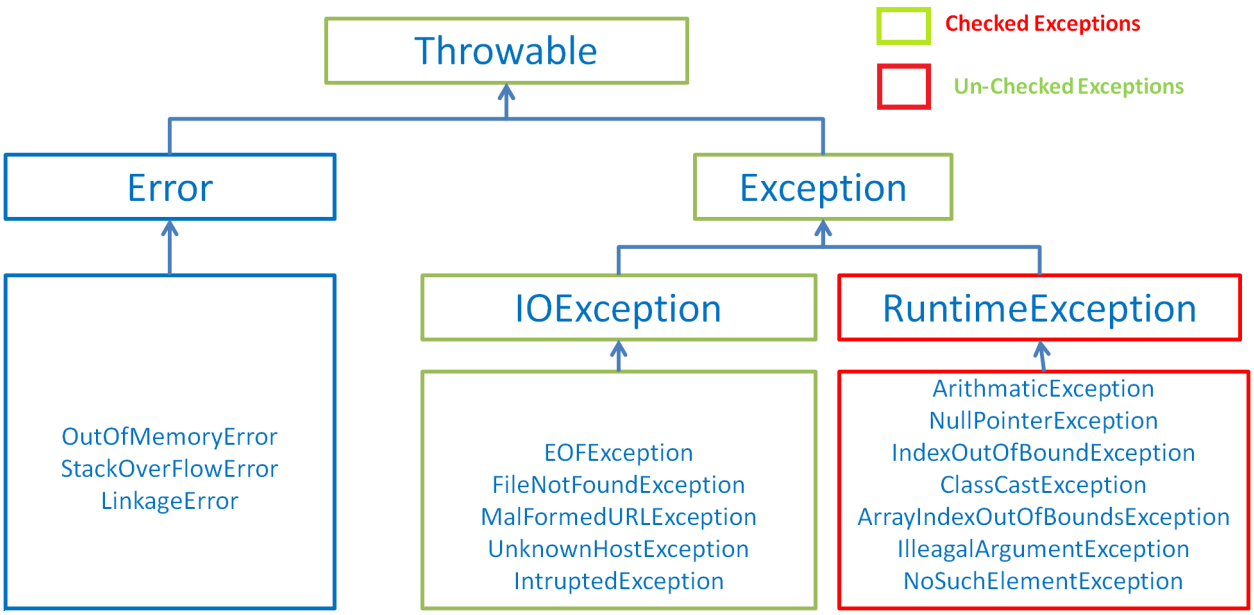
SortedMap

TreeMap:

TreeMap is the implementation of Map and SortedMap. It maintains ascending order.

==========

Exceptions:



Throwable:

The java.lang.Throwable class is the root class of Java Exception hierarchy

Exception: Exception is an abnormal condition.

Exception Types:

****

Checked Exceptions (CompiletimeException):

Checked exceptions are checked at compile-time.

- ✓ IOException
- ✓ ClassNotFoundException
- ✓ SQLException

Un Checked Exceptions (RuntimeException):

Checked exceptions are checked at run-time.

- ✓ NullPointerException

- ✓ ArthemeticException
- ✓ NumberFormatException

IndexOutOfBoundException

- ✓ ArrayIndexOutOfBoundException
- ✓ StringIndexOutOfBoundException

Error:

Error is irrecoverable

- ✓ OutOfMemoryError
- ✓ StackOverFlowError
- ✓ VirtualMachineError

Other things in Exceptions:

---

try:

The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone.

catch:

The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.

throw:

The "throw" keyword is used to throw an exception.

throws:

The "throws" keyword is used to declare exceptions. It doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature.

finally:

The "finally" block is used to execute the important code of the program. It is executed whether an exception is handled or not.

==========

Threads:

A thread is light weight process. It has a separate path of execution

Multi Threading:

Multithreading in Java is a process of executing multiple threads simultaneously.

Advantages of Java Multithreading

---

1) It doesn't block the user because threads are independent and you can perform multiple operations at the same time.

2) You can perform many operations together, so it saves time.

3) Threads are independent, so it doesn't affect other threads if an exception occurs in a single thread.

Multitasking

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved in two ways:

Process-based Multitasking (Multiprocessing)

Thread-based Multitasking (Multithreading)

1) Process-based Multitasking (Multiprocessing)

Each process has an address in memory. In other words, each process allocates a separate memory area.

A process is heavyweight.

Cost of communication between the process is high.

Switching from one process to another requires some time for saving and loading registers, memory maps, updating lists, etc.

2) Thread-based Multitasking (Multithreading)

Threads share the same address space.

A thread is lightweight.

Cost of communication between the thread is low.

Ways to create the threads:

Extends the Thread class

Implementing the Runnable Interface

Thread Life Cycle:

New

Runnable

Running

Waiting

Terminated

Inter thread communication:

----

wait (Causes the current thread to wait until another thread invokes the notify().)

notify (Wakes up a single thread that is waiting on this object's monitor.)

notifyAll (Wakes up all the threads that called wait( ) on the same object.)

Other thread things:

Thread Pool:

Java Thread pool represents a group of worker threads that are waiting for the job and reuse many times.

In case of thread pool, a group of fixed size threads are created. A thread from the thread pool is pulled out and assigned a job by the service provider. After completion of the job, thread is contained in the thread pool again.

Thread Synchronization:

Synchronization in java is the capability to control the access of multiple threads to any shared resource.

Java Synchronization is better option where we want to allow only one thread to access the shared resource.

Types of Synchronization:

There are two types of synchronization

- ✓ Process Synchronization
- ✓ Thread Synchronization

Thread Synchronization:

- ✓ Synchronized method.
- ✓ Synchronized block.
- ✓ Static synchronization.

Sleep:

(Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.),

Join:

(The join() method waits for a thread to die. In other words, it causes the currently running threads to stop executing until the thread it joins with completes its task.)

Yeild:

(Causes the currently running thread to yield to any other threads of the same priority that are waiting to be scheduled.),

Daemon Thread (Its a low priority thread runs in background. Ex: Garbage Collector)

Thread Priorities:

Each thread have a priority. Priorities are represented by a number between 1 and 10. In most cases, thread schedular schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.

MIN_PRIORITY - 1

MAX_PRIORITY - 10

NORM_PRIORITY - 5

ThreadGroup in Java:

Java provides a convenient way to group multiple threads in a single object. In such way, we can suspend, resume or interrupt group of threads by a single method call.

Java Shutdown Hook

The shutdown hook can be used to perform cleanup resource or save the state when JVM shuts down normally or abruptly. Performing clean resource means closing log file, sending some alerts or something else. So if you want to execute some code before JVM shuts down, use shutdown hook.

Deadlock in java:

Deadlock in java is a part of multithreading. Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread. Since, both threads are waiting for each other to release the lock, the condition is called deadlock.

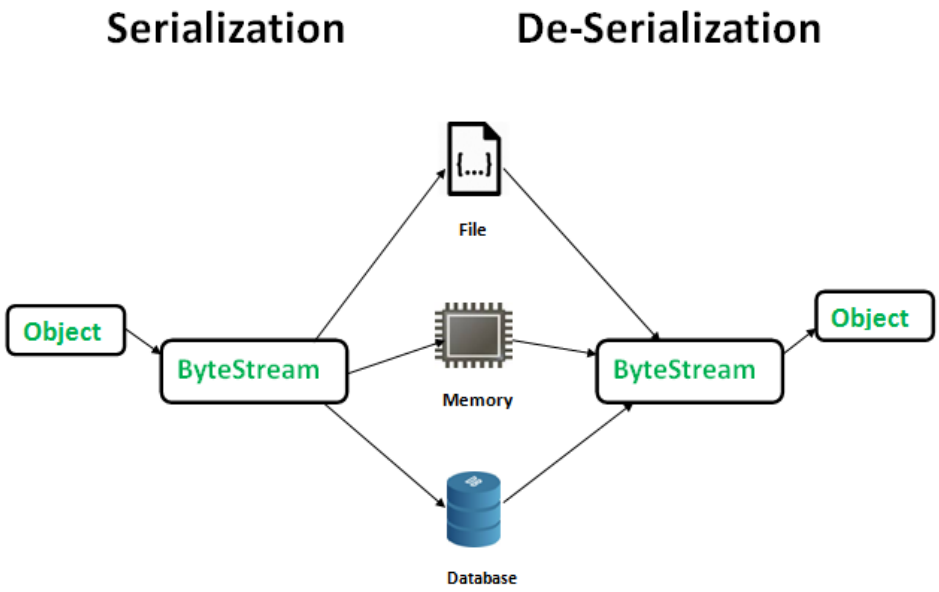
Interrupting a Thread:

If any thread is in sleeping or waiting state (i.e. sleep() or wait() is invoked), calling the interrupt() method on the thread, breaks out the sleeping or waiting state throwing InterruptedException. If the thread is not in the sleeping or waiting state, calling the interrupt() method performs normal behaviour and doesn't interrupt the thread but sets the interrupt flag to true.

Difference between preemptive scheduling and time slicing

Under preemptive scheduling, the highest priority task executes until it enters the waiting or dead states or a higher priority task comes into existence. Under time slicing, a task executes for a predefined slice of time and then reenters the pool of ready tasks. The scheduler then determines which task should execute next, based on priority and other factors.

Serialization & Deserialization & Externilizable:



Serialization:

****

Its a process of writing the state of Objects into the file.

The process of converting the Object to file supported form/network supported from.

Serilization can be implemented with the help of ObjectOutputStream

ex:

//Saving of object in a file

Emp object = new Emp("ab", 20, 2, 1000);

FileOutputStream file = new FileOutputStream(filename);

ObjectOutputStream out = new ObjectOutputStream(file);

// Method for serialization of object

out.writeObject(object);

Here ObjectOutputStream convert the data as bytes and FileOutputStream writes that bytes to file.

Deserialization:

****

Its a process of getting the existing state of an object from file.

The process of converting the file supported form/network supported from to Object from.

DSerilization can be implemented with the help of ObjectInputStream.

Ex:

```
// Reading the object from a file
Emp object = null;
FileInputStream file = new FileInputStream(filename);
ObjectInputStream in = new ObjectInputStream(file);

// Method for deserialization of object
object = (Emp)in.readObject();
```

Here ObjectInputStream convert the bytes as data and FileInputStream writes that data to file.

Transient:

If we dont want to make any of the variable to be serialiazable then will use Transient modifier.

Transient is applicable for varables only not the classes and objects.

If we use the transient jvm will store the default value instead of actual value.

Transient is not applicable for Static varaibles because is transient applicable for instance variables.

Transient is not applicable for Final varaibles because final varibale values is assigned at compiled time.

Can we serialize the multiple objects:

Yes we can serilize multiple objects.

Ex: Cat c = new Cat();

Dog d = new Dog();

Rat r = new Rat();

Here we should read the objects in Objects created order only. First we should get Cat followed by Dog and Rat. If we change the order ofobjects will get Class cast exception.

To avoid the above problem

we need to use instanceof operator.

SerialVersionUID

The serialization at runtime associates with each serializable class a version number, called a serialVersionUID, which is used during deserialization to verify that the sender and receiver of a serialized object have loaded classes for that object that are compatible with respect to serialization.

Why so we use SerialVersionUID : SerialVersionUID is used to ensure that during deserialization the same class (that was used during serialize process) is loaded.

What will happen if we use the default SerialVersionUID for all classes:

When a Serializable class object is serialized Java Runtime associates a serial version no.(called as serialVersionUID) with this serialized object. At the time when you deserialize this serialized object Java Runtime matches the serialVersionUID of serialized object with the serialVersionUID of the class. If both are equal then only it proceeds with the further process of deserialization else throws InvalidClassException.

Externilizable interface

Externalization serves the purpose of custom Serialization, where we can decide what to store in stream.

Externalizable interface present in java.io, is used for Externalization which extends Serializable interface. It consist of two methods which we have to override to write/read object into/from stream which are-

// to read object from stream

void readExternal(ObjectInput in)

// to write object into stream

void writeExternal(ObjectOutput out)

Generics:

Generics will provide the type safe application.

Types of Generics:

Generic Classes:

Like C++, we use <> to specify parameter types in generic class creation. To create objects of generic class, we use following syntax.

```
// We use < > to specify Parameter type
class Test<T>
{
   // An object of type T is declared
   T obj;
   Test(T obj) {  this.obj = obj;  }  // constructor
   public T getObject()  { return this.obj; }
}
```

Generic Methods:

We can also write generic functions that can be called with different types of arguments based on the type of arguments passed to generic method, the compiler handles each method.

```
// A Simple Java program to show working of user defined
// Generic functions

class Test
{
   // A Generic method example
   static <T> void genericDisplay (T element)
   {
      System.out.println(element.getClass().getName() +
                         " = " + element);
   }
```

- ✓ T - Type
- ✓ E - Element
- ✓ K - Key
- ✓ N - Number
- ✓ V - Value

Wildcard in Java Generics

The ? (question mark) symbol represents the wildcard element. It means any type. If we write <? extends Number>, it means any child class of Number, e.g., Integer, Float, and double. Now we can call the method of Number class through any child class object.

We can use a wildcard as a type of a parameter, field, return type, or local variable. However, it is not allowed to use a wildcard as a type argument for a generic method invocation, a generic class instance creation, or a supertype.

Inner Classes:

A class which is declared inside another class is called as inner class.

Inner classes introduced in Java 1.1 version.

Difference between nested class and inner class in Java

Inner class is a part of nested class. Non-static nested classes are known as inner classes.

When we should go for inner classes:

Without existing outer class there is no change of existing inner class.

class car{

class engine {}

}

ex: With out existing car class there is no chance of existing engine class.

Relation b/w outer and inner class is has-a relationship.

Types of inner classes:

---

1. Member inner class.
2. Local inner classes.
3. Ananymous inner classes.
4. Static inner classes.

****

Member inner class:

If we are declaring any named class with out static modifier inside a class is named as member inner class.

Ex 1:

```
class Outer{
class Inner{
}
}
```

After compilation will see two .class files:

Outer.class

Outer$Inner.class

Output:

---

java Outer.class: No such method error

java Outer$Inner.class: No such method error

Ex 2:

```
class Outer{
class Inner{
}
public static void main(String [] args){
sout("Inside outer class main method");
}
}
```

Output:

---

java Outer.class: Inside outer class main method

java Outer$Inner.class: No such method error

Ex 3:

```
class Outer{
class Inner{
public static void main(String [] args){
sout("Inside outer class main method");
}
}
}
```

Output:

---

java Outer.class: Inner class can not have static declaration (Compile time error)

java Outer$Inner.class: Inner class can not have static declaration (Compile time error)

Inside inner class we can not declare any static members hence we can't declare main methos and we can't run inner class directly from command prompt.

Ex 4:

```
class Outer{
class Inner{
public void m1(){
sout("Inside Inner class method");
}
}
public static void main(String [] args){
sout("Inside outer class main method");
Outer.Inner outer = new Outer().new Inner();
outer.m1();
}
}
```

Output:

----

java Outer.class:

Inside outer class main method

Inside Inner class method

java Outer$Inner.class:

Inside outer class main method

Inside Inner class method

Ex 5:

```
class Outer{
class Inner{
public void m1(){
sout("Inside Inner class M1 method");
}
}

public void m2(){
sout("Inside Outer Class M2 method");
Inner inner = new Inner();
inner.m1();
}

public static void main(String [] args){
sout("Inside outer class main method");
Outer outer = new Outer();
outer.m2();
}
}
```

Output:

----

java Outer.class:

Inside outer class main method

nside Outer Class M2 method

Inside Inner class M1 method

java Outer$Inner.class:

Inside outer class main method

inside Outer Class M2 method

Inside Inner class M1 method

Very important for interview:

****

class Outer{

class Inner{}

}

Outer outerRef = new Outer();

Outer.Inner innerRef = new Outer().new Inner();

By using inner class reference (outerRef) we can access only inner classes members and varibales not the outer class members.

By using Outer class reference (innerRef) we can access only outer classes members and varibales not the inner class members.

In java all outer classes members accessble to inner classes members and all inner classes members accessble to outer classes members.

```
class Outer{
int i =10
class Inner{
void m1(){
System.out.println(i);
System.out.println("Inside Inner m1");
}
}
}
```

In general, Inner classes are not allowing static declrations directly, but static keyword allowed along with final keyword.

```
class Outer{
class Inner{
static final int j =20;
}
}
```

We can extend one inner class to another class but both the inner classes must be declared in same outer classes.

```
class Acount{
class A{}
class B extends A{}
}
```

Q) Is it possible to declare an interface inside a class?

==========

Ans) Yes. But, respective implemntation class should be provided in same outer class.

```
class Outer{
interface A{
voild m1();
}
class B implements A{
void m1(){
System.out.println("Implementing the interface methods");
}
}
```

Q) Is it possible to declare an abstract inside a class?

==========

Ans) Yes. But, respective implemntation class should be provided in same outer class.

```
class Outer{
abstract class A{
abstract
voild m1();
}
class B extends A{
void m1(){
System.out.println("Implementing the interface methods");
}
}
```

****
****

Static Inner Classes

==========

Declaring a static classes inside a classe is called as Static Inner Class.

```
class Outer{

public void m1(){
System.out.println("Inside Outer M1");
}

static class Inner{
public void m1(){
System.out.println("Inside Outer M1");
}
}

//Declaring main method inside Outer Class
public static void main(String [] args){
Outer outer = new Outer();
outer.m1();

Outer.Inner inner = new Outer.Inner();
inner.m1();
}

}
```

Important:

Static inner class will allow only static members directly without final keyword.

```
class Outer{
static class Inner{
static int j =20; //No need of final here
}
}
```

Static inner class will allow only static memebers of the Outer class.

```
class Outer{
static int i =10;
int j =20;

public void m1(){
System.out.println("Inside Outer M1");
}

static class Inner{

public void m1(){
System.out.println(i); //10
System.out.println(j); //error
System.out.println("Inside Outer M1");
}
}
```

```
//Declaring main method inside Outer Class
public static void main(String [] args){
Outer outer = new Outer();
outer.m1();

Outer.Inner inner = new Outer.Inner();
inner.m1();
}

}
```

Q: Is it possible to provide an class inside an interface

Ans: Yes

```
interface Outer{
class A // No need of static keyword here as classes defined inside interface by default are static classes
{
void m1();
}

//Declaring main method inside Outer Class
public static void main(String [] args){
Outer outer = new Outer();
outer.m1();

Outer.A inner = new Outer.A();
inner.m1();
```

```
}
}
```

Q: Is it possible to provide an abstract class inside an interface

---

Ans: Yes

```
interface Outer{
abstract class A
{
void m1(){
System.out.println("Inside abstract class M1");
}
abstaract void m2();
}
class B extends A{
void m2(){
System.out.println("Inside class B  M2");
}
}

//Declaring main method inside Outer Class
public static void main(String [] args){
Outer outer = new Outer();
outer.m1();

Outer.A inner = new Outer.B();
inner.m1();
inner.m2();
```

```
}
}
```

Q: Is it possible to provide an interface inside an interface

---

Ans: Yes

```
interface I1{

interface I2{
void m1();
void m2();
}

class inner implements I2{
void m1(){
System.out.println("Inside inner class m1");
}
void m2(){
System.out.println("nside inner class m2");
}
}

//Declaring main method inside Outer Class
public static void main(String [] args){
Outer outer = new Outer();
outer.m1();

I1.I2 inner = new I1.Inner();
```

```
inner.m1();
inner.m2();
}

}
```

Q: Is it possible to provide an class inside an abstract class

---

Ans: Yes

```
abstract class Outer{
class Inner{
        void m1(){
        System.out.println("Inside inner class m1");
        }
}

class ImpClass extends Outer{

}
//Declaring main method inside Outer Class
public static void main(String [] args){
 Outer.Inner inner = new ImpClass().new Inner();
 inner.m1();
}

}
```

Q: Is it possible to provide an abstract class inside an abstract class

---

Ans: Yes

```
abstract class Outer{

abstract class Inner{
void m1(){
        System.out.println("Inside inner class m1");
        }
abstract void m2();
}
class ImpClass extends Inner{
void m2(){
        System.out.println("Inside ImpClass class m2");
        }
}

class ConcClass extends Outer{
}

//Declaring main method inside Outer Class
public static void main(String [] args){
Outer.Inner inner = new ConcClass().new ImpClass();
inner.m1();
inner.m2();
}
}
```

Q: Is it possible to provide an interface inside an abstract class

---

Ans: Yes

```
abstract class Outer{

interface I{
void m1();
}

class B imlements I{
void m1(){
        System.out.println("Inside class B of m1");
        }
}

Class C extends Outer{}

//Declaring main method inside Outer Class
public static void main(String [] args){
 Outer.I inner = new C().new B();
}

}
```

****

Local Inner Class

****

A class which is declared inside a method is known as Local Inner Class.

```
class Outer{

void m1(){

class Inner(){

void m1(){
System.out.println("Inside class Inner of m1");
}

//Creating the Object
Inner inner = new Inner();
inner.m1();

}
}
//Declaring main method inside Outer Class
public static void main(String [] args){
Outer outer = new Outer();
outer.m1();
}

}
```

****

Anonymous Inner Class:

****

A class that does not have any name is called as Anonymous inner classes.

Why we need Anonymous inner class:

This is used to implementation for interfaces and abstract classes.

```
interface A{
void m1();
}

class Outer{
 A a = new A(){
 void m1(){
 System.out.println("Inside Outer class of m1");
 }
};
//Declaring main method inside Outer Class
public static void main(String [] args){
 Outer outer = new Outer();
 outer.a.m1();
}
}
```

Q)For interface we can implement methods using Implementation and for Abstract class we can implement methods using concerete class but why we need Ananymous inner class

---

A) If we want to access only the members of interface/abstract class then will go for Anonymous inner class.

Q) When we can go for Implementation class for interface and Concrete class for Abstract class

---

A) If we want to provide the implementation for available methods in interface/Abstract class along with Implementation and Concrete class specific methods will go for Implementation class and Concrete class.

Note:

When we want to pass interface or abstaract class reference variable as parameter to a method there we can use ananymous inner class as a parameter to method directly with out defining implementation class or sub class.

```
interface I{
        void m1();
        void m2();
}

class X{
        void m3(I i){
        System.out.println("Inside m3 method pass interface");
        i.m1();
        i.m2();
        }
}

class Test{
public static void main(String [] args){
        X x = new X(new I(){
        public void m1(){
        System.out.println("Inside class A method 1");
        }
```

```
        public void m2(){
        System.out.println("Inside class A method 2");
        }
        };);

        x.m3(i);
}
}
```

****

Enum

****

Java Enums are classes that have a fixed set of constants or variables that do not tend to changes. The enumeration in java isa achieved by using enum keyword. The java enum conatants are static and final implicitly.

Why And When To Use Enums?

****

Use enums when you have values that you know aren't going to change, like month days, days, colors, deck of cards, etc.

Advantages Of Enum:

---

1. Enums provide the typesafety.
2. Enum is easily usable in Switch cases
3. Enum can be traveresed.
4. Enum has fields methods and constructors.
5. Enums can implement the interfaces.

****
****

Java I/O streams.

I/O streams provide the classes to store and read the data from files.

Standard/Default streams:

Standard Input − This is used to feed the data to user's program and usually a keyboard is used as standard input stream and represented as System.in.

Standard Output − This is used to output the data produced by the user's program and usually a computer screen is used for standard output stream and represented as System.out.

Standard Error − This is used to output the error data produced by the user's program and usually a computer screen is used for standard error stream and represented as System.err

Types of Streams:

Depending upon the data a stream holds, it can be classified into:

Byte Stream (Byte stream is used to read and write a single byte (8 bits) of data.)

Character Stream (Character stream is used to read and write a single character of data.)

Byte Stream Types:

****

Input Streams:

==========

FileInputStream (The FileInputStream class of the java.io package can be used to read data (in bytes) from files.)

ByteArrayInputStream (The ByteArrayInputStream class of the java.io package can be used to read an array of input data (in bytes).)

ObjectInputStream (The ObjectInputStream class of the java.io package can be used to read objects that were previously written by ObjectOutputStream.)

FilterInputStream

i.BufferedInputStream (The BufferedInputStream class of the java.io package is used with other input streams to read the data (in bytes) more efficiently.)

ii.DataInputStream (Java DataInputStream class allows an application to read primitive data from the input stream)

Output Streams:

==========

FileOutputStream (The FileOutputStream class of the java.io package can be used to write data (in bytes) to the files.)

ByteArrayOututStream (The ByteArrayOutputStream class of the java.io package can be used to write an array of output data (in bytes).)

ObjectOutputStream (The ObjectOutputStream class of the java.io package can be used to write objects that can be read by ObjectInputStream.)

FilterInputStream

i.BufferedOutputStream (The BufferedOutputStream class of the java.io package is used with other output streams to write the data (in bytes) more efficiently.)

ii.DataOutputStream (Java DataOutputStream class allows an application to write primitive data from the input stream)

Character Stream Types:

----

1. Reader

****

BufferedReader (The BufferedReader class of the java.io package can be used with other readers to read data (in characters) more efficiently.)

InputStreamReader (The InputStreamReader class of the java.io package can be used to convert data in bytes into data in characters.)

FileReader (The FileReader class of the java.io package can be used to read data (in characters) from files.)

StringReader (The StringReader class of the java.io package can be used to read data (in characters) from strings.)

2. Writer

****

BufferedWriter (The BufferedWriter class of the java.io package can be used with other writers to write data (in characters) more efficiently.)

OutputStreamWriter (The OutputStreamWriter class of the java.io package can be used to convert data in character form into data in bytes form.)

FileWriter (The FileWriter class of the java.io package can be used to write data (in characters) to files.)

StringWriter (The StringWriter class of the java.io package can be used to write data (in characters) to the string buffer.)

==========

Java Reflections:

****

Java Reflection is a process of examining or modifying the run time behavior of a class at run time.

Examples:

Javac

Javap

java

Above commands will use the Reflections to complie, run and compile the Java programs. Here above will read the variables, methods by using the Reflections API and executing those during the execution time.

Few useful methods of Reflection:

==========

getClass()

****

method is used to get the name of the class to which an object belongs.

getConstructors()

****

method is used to get the public constructors of the class to which an object belongs.

getMethods():

****

method is used to get the public methods of the class to which an objects belongs.

public static Class forName(String className)throws ClassNotFoundException :

****

loads the class and returns the reference of Class class.

3) public Object newInstance()throws InstantiationException,IllegalAccessException:

****

creates new instance.

4) public boolean isInterface():

****

checks if it is interface.

5) public boolean isArray():

****

checks if it is array.

6) public boolean isPrimitive():

****

checks if it is primitive.

7) public Class getSuperclass():

****

returns the superclass class reference.

8) public Field[] getDeclaredFields()throws SecurityException:

****

returns the total number of fields of this class. Used to get the private field. Returns an object of type Field for specified field name.

Field.setAccessible(true) : Allows to access the field irrespective of the access modifier used with the field.

9) public Method[] getDeclaredMethods()throws SecurityException:

****

returns the total number of methods of this class.

10) public Constructor[] getDeclaredConstructors()throws SecurityException:

****

returns the total number of constructors of this class.

11) public Method getDeclaredMethod(String name,Class[] parameterTypes)throws NoSuchMethodException,SecurityException:

****
****

returns the method class instance.

Drawbacks:

****

Performance Overhead:

Because reflection involves types that are dynamically resolved, certain Java virtual machine optimizations can not be performed. Consequently, reflective operations have slower performance than their non-reflective counterparts, and should be avoided in sections of code which are called frequently in performance-sensitive applications.

Security Restrictions:

Reflection requires a runtime permission which may not be present when running under a security manager. This is in an important consideration for code which has to run in a restricted security context, such as in an Applet.

Exposure of Internals:

Since reflection allows code to perform operations that would be illegal in non-reflective code, such as accessing private fields and methods, the use of reflection can result in unexpected side-effects, which may render code dysfunctional and may destroy portability. Reflective code breaks abstractions and therefore may change behavior with upgrades of the platform.